High Performance Software Defined Radio (HPSDR)

John Melton

G0ORX/N6LYT

john.d.melton@googlemail.com
What's Is HPSDR All About?

A group of worldwide enthusiasts interested in developing an open source hardware and software project intended as a "next generation" Software Defined Radio for use by Radio Amateurs and Short Wave Listeners.
Who are TAPR?

Tucson Amateur Packet Radio

TAPR is a community that provides leadership and resources to radio amateurs for the purpose of advancing the radio art.

For HPSDR they typically help with the development of the hardware by funding the cost of building Alpha and Beta boards. They will then make a production run of some number of boards that they will sell through their web site to recoup their costs for the development.
The TAPR Open Hardware License ("OHL") provides a framework for hardware projects that is similar to the one used for Open Source software.

This isn't as straightforward as it seems because legal concepts that work well for software (such as copyright and copyleft) don't neatly fit when dealing with hardware products and the documentation used to create them.

Noncommercial OHL restricts use of design for commercial use.
Software Defined Radio (SDR)

An SDR is a communications device where the typical hardware components such as mixers, filters, amplifiers, modulators/demodulators, detectors, etc. are implemented in software.

Such a design produces a radio which can receive and transmit widely different radio protocols (sometimes referred to as waveforms) based solely on the software used.
Basic Hardware Components

Atlas (Backplane)

Ozy/Magister/Metis (I/O)

Mercury (Rx)

Penelope (Tx)
Firmware Development

Written in VHDL

Quartus II Web Edition FPGA design software

- Free download from Altera (www.altera.com)
  - Windows and Linux versions.
module cic( clock, in_strobe, out_strobe, in_data, out_data );

  //design parameters
  parameter STAGES = 3;
  parameter DECIMATION = 16;
  parameter IN_WIDTH = 18;

  //computed parameters
  //ACC_WIDTH = IN_WIDTH + Ceil(STAGES * Log2(DECIMATION))
  //OUT_WIDTH = IN_WIDTH + Ceil(Log2(DECIMATION) / 2)
  parameter ACC_WIDTH = IN_WIDTH + 12;
  parameter OUT_WIDTH = IN_WIDTH + 2;

  input clock;
  input in_strobe;
  output reg out_strobe;
  input signed [IN_WIDTH-1:0] in_data;
  output signed [OUT_WIDTH-1:0] out_data;

  //----------------------------------------------------------------------------
  //                               control
  //----------------------------------------------------------------------------
  reg [15:0] sample_no;
  initial sample_no = 15'd0;

  always @(posedge clock)
  begin
    if (in_strobe)
      begin
        if (sample_no == (DECIMATION-1))
          begin
            sample_no <= 0;
            out_strobe <= 1;
          end
        else
          begin
            sample_no <= sample_no + 8'd1;
            out_strobe <= 0;
          end
      end
    else
      begin
        sample_no <= sample_no + 8'd1;
        out_strobe <= 0;
      end
  end

  //----------------------------------------------------------------------------
  //                                stages
  //----------------------------------------------------------------------------
  wire signed [ACC_WIDTH-1:0] integrator_data [0:STAGES];
  wire signed [ACC_WIDTH-1:0] comb_data [0:STAGES];

  assign integrator_data[0] = in_data;
  assign comb_data[0] = integrator_data[STAGES];

  genvar i;
  generate
    for (i=0; i<STAGES; i=i+1)
      begin : cic_stages

        cic_integrator #(ACC_WIDTH) cic_integrator_inst(
          .clock(clock),
          .strobe(in_strobe),
          .in_data(integrator_data[i]),
          .out_data(integrator_data[i+1])
        );

        cic_comb #(ACC_WIDTH) cic_comb_inst(
          .clock(clock),
          .strobe(out_strobe),
          .in_data(comb_data[i]),
          .out_data(comb_data[i+1])
        );
      end
  endgenerate

  //----------------------------------------------------------------------------
  //                            output rounding
  //----------------------------------------------------------------------------
  assign out_data = comb_data[STAGES][ACC_WIDTH-1:ACC_WIDTH-OUT_WIDTH] +
                   {{(OUT_WIDTH-1){1'b0}}, comb_data[STAGES][ACC_WIDTH-OUT_WIDTH-1]};
  endmodule
Atlas

Passive backplane.

6 DIN connectors.

20 Pin ATX Power connector.
OZY/Magister

Interface controller.

Cypress FX2 USB 2.0 controller

Altera Cyclone II FPGA.

Interface for PTT and CW paddles.
Metis (OzyII)

Interface controller.

10/100/1000 ethernet.

Altera Cyclone III FPGA.

Interface for PTT and CW paddles.
Both OZY and Magister can also be configured to look like a USB-Blaster for loading FPGA code to Mercury and Penelope.
Mercury

0-65 MHz direct sampling receiver.

Linear Technology LTC2208 130MSPS 16-bit A/D converter.

Altera Cyclone III FPGA.

Digital Down Conversion to 48K, 96K or 192K.
Mercury Performance

ADC overload: -12dBm (preamp on), +8dBm (preamp off)

MDS (500Hz), 160m - 6m: -138dBm (preamp on), -118dBm (preamp off)

IP3 equivalent (independent of spacing): +33dBm (preamp on), >+50dBm (preamp off)
Penelope

½ watt transmitter/exciter board.

Digital Up Conversion.

Altera Cyclone II FPGA.

Microphone input and A to D converter.
Penelope

WSPR code available to download to the FPGA that will allow Penelope to run in a stand alone.

Requires building code with modified call for the user and frequency to transmit on before downloading to Penelope.
Excalibur

External 10MHz reference clock. Built in TCXO that is better than the on board oscillators.
Input for external GPS Disciplined Oscillator.
HPSDR

LPU (Linear Power Supply)

Pennywhistle (16-20 watt PA)

Alex (Filters, Ant switch)
- Single board, Metis, Mercury, Penelope.
Hermes

- Alpha version with filter board (Apollo)
Tayloe Detector (Dan Tayloe)

The switch rotates at the carrier frequency so that each capacitor samples the signal once each revolution. The 0° and 180° capacitors differentially sum to provide the in-phase (I) signal and the 90° and 270° capacitors sum to provide the quadrature (Q) signal.
LT2208 ADC sampling at 125MHz
ADC output  0 – 62.5MHz
Decimate by 640
Output = 125MHz/640 = 195ksp (24 bit samples)
By decimation we have eased the load on the PC but increased the complexity of the DDC.

But there is an additional advantage of decimation!

Every time we decimate by 2 we increase the output SNR by 3dB.
I and Q signals

I – in-phase
Q – quadrature (delayed by 90 degrees)

Use FFT to convert complex I/Q time domain signal to frequency domain. Each bin contains spectral density of the signal within that bin's bandwidth. If we are sampling at 48000 samples per second and have 4096 bins for the FFT then each bin has a bandwidth of 11.71Hz.

Further filtering and processing allows us to demodulate the signals.

Inverse FFT used to convert the frequency domain signal back to time domain

- A Software-Defined Radio for the Masses, Part 1

- Quadrature Signals: Complex, But Not Complicated
  http://www.dspguru.com/dsp/tutorials/quadrature-signals
DttSP

Open Source Digital Signal Processing package developed by Dr. Frank Brickle and Dr. Robert McGwier (uses FFTW3).

*Implements the basic modulation, demodulation, signal conditioning, and synchronisation processes required to operate a high performance transceiver using DSP as the detection and synthesis stages.*
HPSDR

Windows Software – FlexRadio PowerSDR
HPSDR

Windows Software – KISS Konsole
HPSDR

Linux Software - ghpsdr
HPSDR

Software

TCP connection for Commands
I/Q data sent over UDP
USB Interface

USB Output Data Stream

USB data is transferred in 512 byte packets.
Always at 48K.

8 bytes of header data (rx freq, tx freq, mox, duplex).

| Sync | Sync | Sync | C0 | C1 | C2 | C3 | C4 |

63 Transmit I/Q data and Audio samples.

<table>
<thead>
<tr>
<th>Tx</th>
<th>Audio</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Q</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
</tr>
</tbody>
</table>

I and Q samples are 16 bits.
Audio samples are 16 bits.
• Software

USB I/Q Data Stream

USB data is transferred in 512 byte packets.

8 bytes of header data.

Receiver I/Q data and Mic sample.

I and Q samples are 24 bits (3 bytes).
Microphone samples are 16 bits (2 bytes).
**Software**

Bandwidth Requirements

- 24 bit I samples
- 24 bit Q samples

- 48000 samples per second = **2304000** bits per second
- 96000 samples per second = **4608000** bits per second
- 192000 samples per second = **9216000** bits per second

For just 1 receiver!

- 8 receivers at 192000 = **73728000** bits per second

My broadband uplink speed is 360 Kbps
Software
Software

![HPSDR Software Screenshot](image-url)
**HPSDR**

- **Software**

  ![Diagram of HPSDR](image)

  - **HPSDR Server**
    - **USB I/Q samples**
    - **Control & Audio**

  - **DSP Server (RX-0)**
    - **RX 0 Buffer**
    - **commands**
    - **RX-0 I/Q**

  - **ghpsdr (RX-n)**
    - **RX n Buffer**
    - **commands**
    - **RX-n I/Q**

  - **Internet Client**
    - **commands**
    - **Spectrum Data**
    - **Audio data**
Software

- Bandwidth requirements per receiver

  - Spectrum data sent as 8 bit values
  - Client requests number of samples - currently 480 at 10 requests per second.

  - Audio data sent as 8-bit aLaw at 8000 samples per second.

  - Spectrum data = 480*10*8 = 34800 bits per second
  - Audio data =8000*8 = 64000 bits per second

  - 98800 bits per second per receiver
Software

Java Applet Client
Runs in any browser window.
Can be run as a standalone application.
Many accesses from around the world to my web page
Software - iPhone client
HPSDR
Software - Android Client
HPSDR

Software - QtRadio

Uses Qt 4 (from Nokia) – multiplatform GUI support
HPSDR
Software – GNU Radio
Demonstration?
HPSDR References

http://openhpsdr.org
  Wiki
  Teamspeak
  Reflector
  Links

Teamspeak recordings
  http://www.hamsdr.com/dnld.aspx

http://www.tapr.org

svn://64.245.179.219/svn/repos_sdr_hpsdr/trunk/N6LYT

g0orx.blogspot.com

john.d.melton@googlemail.com